Why Dependent Types Matter

Thorsten Altenkirch

School of Computer Science
University of Nottingham

October 5, 2011

The Greek-ASCII dichotomy

- Programs are (were ?) written in ASCII . . .
- Papers in theoretical Computer Science use greek letters . . .

- Programmers don't do proofs . . .
- Theoreticians don't write programs . . .

- Can we bridge the gap?

Another observation

- Compilers don’t read comments . . .
- Sometimes they should!
- How can we make informal statements formal
- and checkable by our software tools?

A clarification

- Formal specifications cannot be the starting point of software development.
- The early stages are exploratory steps involving prototypes
- In the beginning we don't know much about the software we are developing.
- Exploratory steps / Consolidation steps.
- Specifications are one of the outputs of consolidation steps.
- Would like to guarantuee that specifications and code fit together.

Proofs

- Unrealistic to hope that all relevant properties are decidable.
- Need proofs as formal objects which provide evidence that an assertion holds.
- Replace oracles
  (decision procedures which answer either yes or no) ...
- ... with evidence producing decision procedures.
- Potential of an economy of proofs
  (who is too blame?).

But how do we do it?

Programming Language + Logic

- Separation of language for programming and reasoning
- Possible for (almost) any programming language
- Conventional logic (1st order, classical)
- Geared to posthoc verification

Dependently Typed Programming

- Functional language with an expressive type system
- Reasoning emerges due to the Curry-Howard principle
- Intuitionistic logic
- Integration of reasoning and programming

From Per to Ulf

Per Martin-Löf

Introduced Type Theory
As a new constructive foundation of Mathematics
since the mid 1970ies

Ulf Norell

Implemented the current Agda system
A functional programming language
and an interactive proof assistant
based on Type Theory
in his PhD in 2005

Rest of the talk

- A taste of Agda
- The Curry-Howard Principle
- Classical logic
- Recursion and induction
- Families of types
- Coinduction
- Design challenges

A taste of Agda

Safe lookup

- Define an operation which extracts the nth element out of a list.

$$\_\,!!\_ : \mathit{List}\ A \to \mathbb{N} \to A$$
$$xs\ !!\ n = ?$$

1st attempt

$$
\begin{array}{l}
\_\,!!\_ : \mathit{List}\ A \to \mathbb{N} \to A \\
[\,]\ !!\ n = \ ? \\
(x :: xs)\ !!\ \mathit{zero} \quad = x \\
(x :: xs)\ !!\ \mathit{suc}\ n = xs\ !!\ n
\end{array}
$$

- We cannot complete this program.
- Agda only allows complete pattern.
- *A* could be empty.

2nd attempt (use monad)

$$
\begin{aligned}
&\_\,!!\_ : \mathit{List}\ A \to \mathbb{N} \to \mathit{Maybe}\ A\\
&[\,]\ !!\ n = \mathit{nothing}\\
&(a :: as)\ !!\ \mathit{zero} = \mathit{just}\ a\\
&(a :: as)\ !!\ \mathit{suc}\ n = as\ !!\ n
\end{aligned}
$$

- We use the *Maybe* monad.
- In Haskell (and other languages) this is built-in.
- Runtime errors may arise at any time.

From *Nat* and *List*

$$
\begin{array}{l}
\textbf{data}\ \mathbb{N} : \mathit{Set}\ \textbf{where} \\
\quad \mathit{zero} : \mathbb{N} \\
\quad \mathit{suc} : (n : \mathbb{N}) \to \mathbb{N}
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{List}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
\quad [\,] : \mathit{List}\ A \\
\quad \_ :: \_ : (x : A)\ (xs : \mathit{List}\ A) \to \mathit{List}\ A
\end{array}
$$

To *Fin* and *Vec*

$$
\begin{array}{l}
\textbf{data}\ \mathit{Fin} : \mathbb{N} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{zero} : \mathit{Fin}\ (\mathit{suc}\ n) \\
\quad \mathit{suc} : (i : \mathit{Fin}\ n) \to \mathit{Fin}\ (\mathit{suc}\ n)
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{Vec}\ (A : \mathit{Set}) : \mathbb{N} \to \mathit{Set}\ a\ \textbf{where} \\
\quad [\,] : \mathit{Vec}\ A\ \mathit{zero} \\
\quad \_ :: \_ : (x : A)\ (\mathit{xs} : \mathit{Vec}\ A\ n) \to \mathit{Vec}\ A\ (\mathit{suc}\ n)
\end{array}
$$

3rd attempt (use dependent types)

$$
\begin{array}{l}
\_\,!!\_\, : \mathit{Vec}\ A\ n \rightarrow \mathit{Fin}\ n \rightarrow A \\
[\,]\ !!\ () \\
(x :: xs)\ !!\ \mathit{zero} = x \\
(x :: xs)\ !!\ \mathit{suc}\ i = xs\ !!\ i
\end{array}
$$

- We have replaced *List* with *Vec* and *Nat* with *Fin*.
- No runtime errors.

- Using dependent types we can eliminate runtime errors
- But what if we read the index from external sources?
- We need to check but only once.
- Runtime errors are clearly localized.

The Curry-Howard principle

The Curry-Howard principle

- We can express certain constraints using dependent types.
- What are the limits of this technology?
- We can encode any logical formula as a dependent type.
- We assign to a logical formula the set of its proofs.

$$
\begin{aligned}
&\mathit{prop} : \mathit{Set}_1 \\
&\mathit{prop} = \mathit{Set}
\end{aligned}
$$

- Proving = constructing a program of this type.

Propositional Logic

Implication $P \to Q$ is given by the type of functions from P to Q.

Conjunction $P \wedge Q$ is given by the type of pairs of elements of P and Q.

$$
\begin{array}{l}
\textbf{data}\ \_\wedge\_\ (P\ Q : \mathit{prop}) : \mathit{prop}\ \textbf{where} \\
\quad \_,\_ : P \to Q \to P \wedge Q
\end{array}
$$

Disjunction $P \vee Q$ is given by the disjoint union of elements of P and Q.

$$
\begin{array}{l}
\textbf{data}\ \_\vee\_\ (P\ Q : \mathit{prop}) : \mathit{prop}\ \textbf{where} \\
\quad \mathit{left} : P \to P \vee Q \\
\quad \mathit{right} : Q \to P \vee Q
\end{array}
$$

How to prove ?

$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$?

Write a program!

$$
\begin{aligned}
&\mathit{distrib} : P \wedge (Q \vee R) \to (P \wedge Q) \vee (P \wedge R) \\
&\mathit{distrib}\ (p, \mathit{left}\ q) = \mathit{left}\ (p, q) \\
&\mathit{distrib}\ (p, \mathit{right}\ r) = \mathit{right}\ (p, r)
\end{aligned}
$$

- Observe that the program is invertible.
- Hence we can prove $\Leftrightarrow$.
- This provides a different explanation than the truth table.
- More accessible to programmers?!

Predicate logic

universal quantification The set of proofs of $\forall x : A.P\,x$ is the set of dependent function $(x : A) \to P\ x$.

existential quantification The set of proofs of $\exists x : A.P\,x$ is the set of dependent pairs:

$$
\begin{array}{l}
\textbf{data}\ \exists\ (A : \mathit{Set})\ (P : A \to \mathit{prop}) : \mathit{prop}\ \textbf{where} \\
\quad \_,\_ : (a : A) \to P\ a \to \exists A\ P
\end{array}
$$

How to prove?

$$\forall x : A.P\,x \to Q \iff (\exists x : A.P\,x) \to Q$$

Write a program!

$$\begin{array}{l} \mathit{curry} : ((\exists A\ P) \to Q) \to (a : A) \to P\ a \to Q \\ \mathit{curry}\ x = \lambda\ a\ x' \to x\ (a, x') \\ \mathit{curry}' : ((a : A) \to P\ a \to Q) \to ((\exists A\ P) \to Q) \\ \mathit{curry}'\ x\ (a, y) = x\ a\ y \end{array}$$

- Generalized form of currying.
  $(P \land Q \to R) \Leftrightarrow (P \to Q \to R)$
- Not just a logical equivalence ...
- but an isomorphism.
- Not all equivalences are isomorphisms.

Classical logic

What about the excluded middle ?

- We cannot prove:

  $$tnd : \{P : prop\} \to P \vee \neg P$$

  and other classical principles.
- Because our logic is intuitionistic and constructive.

The classical Babelfish 

Classical reasoner says:	Babelfish translates to:
$A \vee B$	$\neg(\neg A \wedge \neg B)$
$\exists x : S.Px$	$\neg\forall x : S.\neg Px$

- *Negative translation*
- $A \vee \neg A$ is translated to $\neg(\neg A \wedge \neg\neg A)$ which is constructively provable.
- A classical reasoner is somebody who is unable to say anything positive.
- However, while the axiom of choice is provable (easily)

$$(\forall a : A.\exists b : B.R\,a\,b) \to \exists f : A \to B.\forall a : A.R\,a\,(f\,a)$$

- its translation is not:

$$(\forall a : A.\neg\forall b : B.\neg R\,a\,b) \to \neg\forall f : A \to B.\neg\forall a : A.R\,a\,(f\,a)$$

Recursion and induction

How to prove ?

$$\forall i\, j\, k : \mathbb{N}.(i + j) + k = i + (j + k)?$$

where

$$\begin{aligned}
_ + _ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
zero + n &= n \\
suc\ m + n &= suc\ (m + n)
\end{aligned}$$

Equality

The only proof that $a = b$ is *refl* if a and b are identical.

$$
\begin{array}{l}
\textbf{data}\ \_ \equiv \_\ (x : A) : A \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{refl} : x \equiv x
\end{array}
$$

We can prove that every function respects equality using pattern matching:

$$
\begin{array}{l}
\mathit{cong} : (f : A \to B)\ \{a\ b : A\} \to a \equiv b \to f\ a \equiv f\ b \\
\mathit{cong}\ f\ \mathit{refl} = \mathit{refl}
\end{array}
$$

Write a program!

$$
\begin{aligned}
&\mathit{assoc} : (i\ j\ k : \mathbb{N}) \to (i + j) + k \equiv i + (j + k) \\
&\mathit{assoc}\ \mathit{zero}\ j\ k = \mathit{refl} \\
&\mathit{assoc}\ (\mathit{suc}\ i)\ j\ k = \mathit{cong}\ \mathit{suc}\ (\mathit{assoc}\ i\ j\ k)
\end{aligned}
$$

- This is a recursive program!
- Induction = primitive recursion
- What is the result of *assoc* $2\ 7\ 3$?

Proof irrelevance

- Indeed *assoc* always returns *refl*.
- There is no point in running *assoc*.
- However, it is important to know that it exists.
- Is this always the case?

Deciding equality

- Equality for 1st order datatypes (like $\mathbb{N}$) is decidable.
- This is witnessed by a boolean function:

$$
\begin{aligned}
&\_ \equiv_? \_ : \mathbb{N} \to \mathbb{N} \to \mathit{Bool} \\
&\mathit{zero} \equiv_? \mathit{zero} = \mathit{true} \\
&\mathit{zero} \equiv_? \mathit{suc}\ n = \mathit{false} \\
&\mathit{suc}\ n \equiv_? \mathit{zero} = \mathit{false} \\
&\mathit{suc}\ n \equiv_? \mathit{suc}\ m = n \equiv_? m
\end{aligned}
$$

- How do we know that this function decides equality?

Decidability

- To decide a proposition means we can show there is a proof ...
- or there cannot be one.

data *Dec* $(P : Set) : Set$ **where**
  $yes : (p : P) \to Dec\ P$
  $no : (\neg p : \neg P) \to Dec\ P$

- A predicate is *decidable*, if each instance can be decided.
- To say that equality is decidable amounts to $(m\ n : \mathbb{N}) \to Dec\ (m \equiv n)$

Deciding equality . . .

$$
\begin{array}{l}
\_ \equiv_? \_ : (m\ n : \mathbb{N}) \to \mathit{Dec}\ (m \equiv n) \\
\mathit{zero} \equiv_? \mathit{zero} = \mathit{yes}\ \mathit{refl} \\
\mathit{zero} \equiv_? \mathit{suc}\ n = \mathit{no}\ (\lambda\ ()) \\
\mathit{suc}\ n \equiv_? \mathit{zero} = \mathit{no}\ (\lambda\ ()) \\
\mathit{suc}\ n \equiv_? \mathit{suc}\ m\ \mathit{with}\ n \equiv_? m \\
\mathit{suc}\ n \equiv_? \mathit{suc}\ m \mid \mathit{yes}\ p = \mathit{yes}\ (\mathit{cong}\ \mathit{suc}\ p) \\
\mathit{suc}\ n \equiv_? \mathit{suc}\ m \mid \mathit{no}\ \mathit{np} = \\
\quad \mathit{no}\ (\lambda\ q \to \mathit{np}\ (\mathit{cong}\ \mathit{pred}\ q))
\end{array}
$$

- Similar structure as the boolean function.
- Instead of returning *true* or *false* . . .
- $\equiv_?$ returns *yes* or *no* and evidence that this is the correct answer.
- Indeed $\equiv_?$'s type already completely specifies its behaviour.

Families of types

How to define $\_ \leqslant \_$?

data $\_ \leqslant \_ : \mathbb{N} \to \mathbb{N} \to \mathit{Set}$ **where**
  $\mathit{le0} : \mathit{zero} \leqslant n$
  $\mathit{leS} : m \leqslant n \to \mathit{suc}\ m \leqslant \mathit{suc}\ n$

- $m \leqslant n$ is the set of derivation trees showing that m is less or equal n.
- E.g. $\mathit{leS}\ (\mathit{leS}\ \mathit{le0}) : 2 \leqslant 4$
- How to prove transitivity?

Write a program!

$$
\begin{aligned}
&\textbf{data}\ \_\leqslant\_ : \mathbb{N} \to \mathbb{N} \to \mathit{Set}\ \textbf{where}\\
&\quad \mathit{le0} : \mathit{zero} \leqslant n\\
&\quad \mathit{leS} : m \leqslant n \to \mathit{suc}\ m \leqslant \mathit{suc}\ n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{trans} : \forall \{l\ m\ n\} \to l \leqslant m \to m \leqslant n \to l \leqslant n\\
&\mathit{trans}\ \mathit{le0}\ p = \mathit{le0}\\
&\mathit{trans}\ (\mathit{leS}\ p)\ (\mathit{leS}\ q) = \mathit{leS}\ (\mathit{trans}\ p\ q)
\end{aligned}
$$

How to define provability?

$$
\begin{array}{l}
\textbf{data}\ \_ \vdash \_ : \mathit{Context} \to \mathit{Formula} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{ass} : \Gamma \cdot A \vdash A \\
\quad \mathit{weak} : \Gamma \vdash A \to \Gamma \cdot B \vdash A \\
\quad \mathit{app} : \Gamma \vdash A \Rightarrow B \to \Gamma \vdash A \to \Gamma \vdash B \\
\quad \mathit{abs} : \Gamma \cdot A \vdash B \to \Gamma \vdash A \Rightarrow B
\end{array}
$$

- Minimal propositional logic.
- $\Gamma \vdash A$ is the set of derivation trees proving A from Γ.
- This is a natural deduction style definition.
- Corresponds to typed λ-calculus with de Bruijn variables.
- Define typed terms directly, not untyped terms and typing relation.

Combinatory logic

$$
\begin{array}{l}
\textbf{data}\ \_\vdash sk\_ : \mathit{Context} \to \mathit{Formula} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{ass} : \Gamma \cdot A \vdash sk\ A \\
\quad \mathit{weak} : \Gamma \vdash sk\ A \to \Gamma \cdot B \vdash sk\ A \\
\quad \mathit{app} : \Gamma \vdash sk\ A \Rightarrow B \to \Gamma \vdash sk\ A \to \Gamma \vdash sk\ B \\
\quad K : \Gamma \vdash sk\ A \Rightarrow B \Rightarrow A \\
\quad S : \Gamma \vdash sk\ (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C
\end{array}
$$

- Can prove equivalence
  $\Gamma \vdash A \Leftrightarrow \Gamma \vdash sk\ A$
- by recursion / induction over derivation trees.
- Key lemma: $\Gamma \cdot A \vdash sk\ B \to \Gamma \vdash sk\ A \Rightarrow B$

Coinduction

Streams

- While *List A* represents the set of finite sequences.
- *Stream A* is the set of infinite sequences.

$$\textbf{data}\ \mathit{Stream}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where}$$
$$\_ :: \_ : A \to \infty(\mathit{Stream}\ A) \to \mathit{Stream}\ A$$

- To define *Stream A* we exploit the notion of lifted types ∞A.
- Delay : $\sharp : A \to \infty A$
- Force : $\flat : \infty A \to A$

Computations on streams

- Define the sequence of numbers starting wth n:

  $$\begin{aligned}&\mathit{from} : \mathbb{N} \to \mathit{Stream}\ \mathbb{N}\\ &\mathit{from}\ n = n :: \sharp(\mathit{from}\ (\mathit{suc}\ n))\end{aligned}$$

- Can we prove?
  $\mathit{mapStream}\ \mathit{suc}\ (\mathit{from}\ n) \approx \mathit{from}\ (\mathit{suc}\ n)$ where

  $$\begin{aligned}&\mathit{mapStream} : (A \to B) \to \mathit{Stream}\ A \to \mathit{Stream}\ B\\ &\mathit{mapStream}\ f\ (a :: as) = f\ a :: \sharp(\mathit{mapStream}\ f\ (\flat as))\end{aligned}$$

Infinite proofs

- Since proofs = programs
- proofs over infinite datastructures
- can be infinite datastructures themselves.
- Extensional equality of streams (bisimilarity).

$$\textbf{data}\ \_\approx\_\ \{A\} : (\mathit{xs}\ \mathit{ys} : \mathit{Stream}\ A) \to \mathit{Set}\ \textbf{where}$$
$$\_::\_ : \forall\, x\ \{\mathit{xs}\ \mathit{ys}\}\ (\mathit{xs}\approx : \infty\, (\flat\mathit{xs} \approx \flat\mathit{ys})) \to x :: \mathit{xs} \approx x :: \mathit{ys}$$

- Can construct an infinite proof:

$$\mathit{nthLem} : (n : \mathbb{N}) \to \mathit{mapStream}\ \mathit{suc}\ (\mathit{from}\ n) \approx \mathit{from}\ (\mathit{suc}\ n)$$
$$\mathit{nthLem}\ n = \mathit{suc}\ n :: \sharp\mathit{nthLem}\ (\mathit{suc}\ n)$$

Design challenges

Termination checking

- Need to ensure programs are total.
- Agda termination checker verifies structural recursion / guardedness .
- Non-structural / non-guarded total programs can be implemented . . .
- . . . but the effort is considerable.
- Need extensible but safe termination checker.
- Reduction to total core language instead of external checker?

Efficient implementation of IDEs

- Interactive program development creates new challenges.
- Symbolic evaluation.
- Typechecking incomplete programs.
- Issues with scaling to larger sized developments.
- Agda: problems with records due to η-expansion.

Efficent compilation

- Naive compilation creates considerable overhead.
- Many expressions don't need to be computed, no computational content.
- See Edwin Brady's work on compilation of dependently typed languages.
- Dependent type provide ample opportunities for novel optimisations (e.g. exploiting finiteness)

Interfacing the real world

- Monads provide a clear interface to effectful programming.
- Haskell's IO monad is opaque.
- How to reason about it?
- What happes when I/O expression appear in dependent types?
- See wouter Swierstra's work on functional specification of I/O.

Proof automatisation

- Want to create proofs (semi) automatically.
- Instead of providing a tactic language ...
- exploit reflection!
- Use Agda to write tactics.
- E.g. see the recent work of Struth and Foster.

Reusability

- Finer types
- reduce reusability
- E.g. instead of lists we have vectors, sorted lists, contexts, ...
- Hard to implement a useful library.
- Use generic programming to derive datatypes
- and share common structure
- Topic of an ongoing research project (Nottingham, Oxford, Strathclyde)

Tricky datatypes

- Agda allows very flexible mutual definitions.
- induction-recursion.
- induction-induction.
- which are not well understood semantically.
- Topic of an ongoing research project (Nottingham, Swansea, Strathclyde).

Extensionality

- The principle of extensionality is not provable in Agda
  $ext : (f\ g : A \to B) \to ((a : A) \to f\ a \equiv g\ a) \to f \equiv g$
- Lack of quotient types.
- New proposal: identify types upto isomorphism (Voevodsky)
- Don't want to add axioms
- because they destroy the computational structure of the theory.
- Can these principles be eliminated?

Conclusions

Conclusions

- DTP: new perspective on certified program development.
- Reasoning emerges from a rich type discipline.
- Covers the whole spectrum from programming to verification.
- Allows a pay-as-you go approach to certification.
- New challenges . . .
- . . . but many of them seem to be unavoidable.